tts: A SAT-Solver for Small, Difficult Instances

Spence, I. (2008). tts: A SAT-Solver for Small, Difficult Instances. *Journal on Satisfiability, Boolean Modeling and Computation*, *4*(null), 173-190.

Published in:
Journal on Satisfiability, Boolean Modeling and Computation

Journal on Satisfiability, Boolean Modeling and Computation 4 (2008) 173-190

tts: A SAT-Solver for Small, Difficult Instances

Ivor Spence i.spence@qub.ac.uk
Institute of Electronics, Communications and Information Technology,
Queen's University Belfast,
Northern Ireland

Abstract

The *Ternary Tree Solver* (`tts`) is a complete solver for propositional satisfiability which was designed to have good performance on the most difficult small instances. It uses a static ternary tree data structure to represent the simplified proposition under all permissible partial assignments and maintains a database of derived propositions known to be unsatisfiable. In the SAT2007 competition version 4.0 won the silver medal for the category `handmade`, speciality UNSAT solvers and was the top qualifier for the second stage for handmade benchmarks, solving 11 benchmarks which were not solved by any other entrant. We describe the methods used by the solver and analyse the competition Phase 1 results on small benchmarks. We propose a first version of a comprehensive suite of small difficult satisfiability benchmarks (`sdsb`) and compare the worst-case performance of the competition medallists on these benchmarks.

KEYWORDS: *SAT-solver, difficult instance, variable ordering, simulated annealing, clause memoisation*

Submitted September 2007; revised January 2007; published June 2008

1. Introduction

The Ternary Tree Solver (`tts`) is a complete, deterministic solver for CNF satisfiability. `tts` cannot compete with state-of-the-art solvers of large industrial and random benchmarks but appears to have good performance on hand-crafted benchmarks (such as hgen8, pigeon-hole, xor-chain etc.) that others find difficult. An extended version (`ttsp`) can generate proofs that benchmarks are not satisfiable.

This article describes the operation of version 4.0. The solver is based on the well-known Davis-Putnam-Logemann-Loveland (DPLL) algorithm[5] and has three main phases: Preliminary stages, Tree building, and Tree walking.

During the preliminary stages unit clause propagation is first carried out. Also, if all occurrences of a particular variable are of the same sign it is safe to assign the corresponding value to the variable and delete any clauses containing it (the pure literal [4] rule). Any clause containing both a variable and its negation is a tautology and is deleted.

Then, if possible, the problem is partitioned into disjoint sub-problems such that no clause spans more than one sub-problem. The rest of the algorithm processes each sub-problem separately, and there is a final stage to merge the separate solutions. For most benchmarks of interest no such partitioning is possible.

The variable ordering stage consists of constructing a linear arrangement of the variables in which, so far as possible, variables which occur in the same clause are close in the arrangement. This is analogous to the Minimum Linear Arrangement problem for hypergraphs [12].

The core of the algorithm then consists of applying partial assignments in an attempt to reduce the proposition to *True* or *False*. This is analogous to DPLL as used in other solvers, except that there the efficiency comes from the good choice of which variable to assign next, whereas in `tts` the efficiency comes from

- the use of a pre-built ternary tree to encapsulate the subsequent partial assignments; and
- the way in which previous goals which were found to be unsatisfiable are remembered so that early pruning of the search tree can be achieved. This has the same aim as clause learning[3], but here each goal is represented as a fragment of the original proposition rather than as a newly generated clause.

In Section 2 we describe the DPLL algorithm. Section 3 describes the preliminary stages for `tts` and in Section 4 the main parts of the algorithm are explained, namely tree building and walking, and clause memoisation. Section 5 shows how the algorithm can be modified to generate resolution-based proofs of unsatisfiability. In Section 6 `tts` is compared with the other medallists from the SAT2007 competition. The comparisons are based firstly on the published results from the competition and then on an extended set of small but difficult benchmarks. Using the worst-case measure described in the results sections, `tts` gives the best results in both cases, although using average-case measures of time or space it is not competitive in the *random* or *industrial* categories of the competition. Section 7 presents the conclusions.

2. The DPLL Algorithm

We include a brief discussion of DPLL[5] because it provides a good starting point for understanding `tts`. Given that there are n variables, the sets U (variables), L (literals), C (clauses) and P (propositions) are defined in Conjunctive Normal Form (CNF) by

$$
\begin{array}{lll}
U & = \{u_i \cdot i \in \{1..n\}\} & \\
L & = \{u \cdot u \in U\} \cup \{\overline{u} \cdot u \in U\} & \\
C & = \mathcal{P}(L) & \text{i.e. power set of } L\text{, so a clause is a set of literals} \\
P & = \mathcal{P}(C) & \text{and a proposition is a set of clauses}
\end{array}
$$

We shall use u and $\overline{u}$ to denote positive and negative literals, and l to denote a literal whose sign is unspecified. ~ complements a literal, so that $\sim u = \overline{u}$ and $\sim\overline{u} = u$.

We use T to denote *True* and F to denote *False*. An assignment a is then a mapping from L to $\{T, F\}$ such that $a(l) = \overline{a(\sim l)}$. A clause c is *interpreted* to have a value T (written $c[a]$) under an assignment a iff at least one literal in the clause is mapped to T. We write

$$c[a] \triangleq \exists l \in c \cdot a(l)$$

which means that the empty clause has the value F under every assignment. Similarly, a proposition p is interpreted to have a value T under an assignment a (written $p[a]$) iff every

clause in p interprets to T. Thus

$$p[a] \triangleq \forall c \in p \cdot c[a]$$

and the proposition with no clauses has the value T under every assignment.

The assignment of T or F to a single variable u can be represented by the corresponding literal (u or $\overline{u}$) and the DPLL method is built around reducing a proposition by making a series of such assignments. Any clause containing the literal is satisfied and so deleted, and any clause containing the negation of the literal cannot be satisfied by this variable so it is simplified by deleting that occurrence. Setting u to T simplifies p to $p|_u$ and setting u to F simplifies p to $p|_{\overline{u}}$ where

$$p|_l = \{c \backslash \{\tilde{l}\} \cdot c \in p \wedge l \notin c\}$$

The algorithm to determine whether there is an assignment a for which $p[a] = T$ is given in Figure 1.

```
boolean DPLL (Proposition p)
{
    if p = ∅
        return T;
    else if ∅ ∈ p
        return F;
    else
        u = a currently unassigned variable;(see below)
        return DPLL (p|u) or DPLL(p|ū);
}
```

Figure 1. The DPLL algorithm

This corresponds to a binary tree of calls to DPLL, where each leaf node n is either $n = \emptyset$ or $\emptyset \in n$. If there is a node $n = \emptyset$ the algorithm return T, otherwise it returns F.

Consider Figure 2 which represents potential traces of the algorithm when applied to the proposition $(A \vee B \vee C) \wedge (A \vee B \vee \overline{C}) \wedge (\overline{A} \vee C) \wedge (\overline{B} \vee C) \wedge (\overline{B} \vee \overline{C})$. The path representing the assignment $\{A, \overline{B}, C\}$ is highlighted using solid lines. This assignment satisfies the proposition and it can be seen that the corresponding path reaches a T leaf. The number of levels in the tree is $1 + n$ and so the algorithm at first sight seems $O(2^n)$. The way to obtain good execution times in practice is through good choices of the next variable to assign, so that either a satisfying assignment is found as early as possible or branches are pruned as early as possible.

We now describe the `tts` solver.

3. Preliminary stages

A number of known simplifications are carried out before the main algorithm. For the small, difficult benchmarks described in the Section 6 these are not usually of any benefit and so they are not of particular interest for this paper, but for some larger benchmarks in

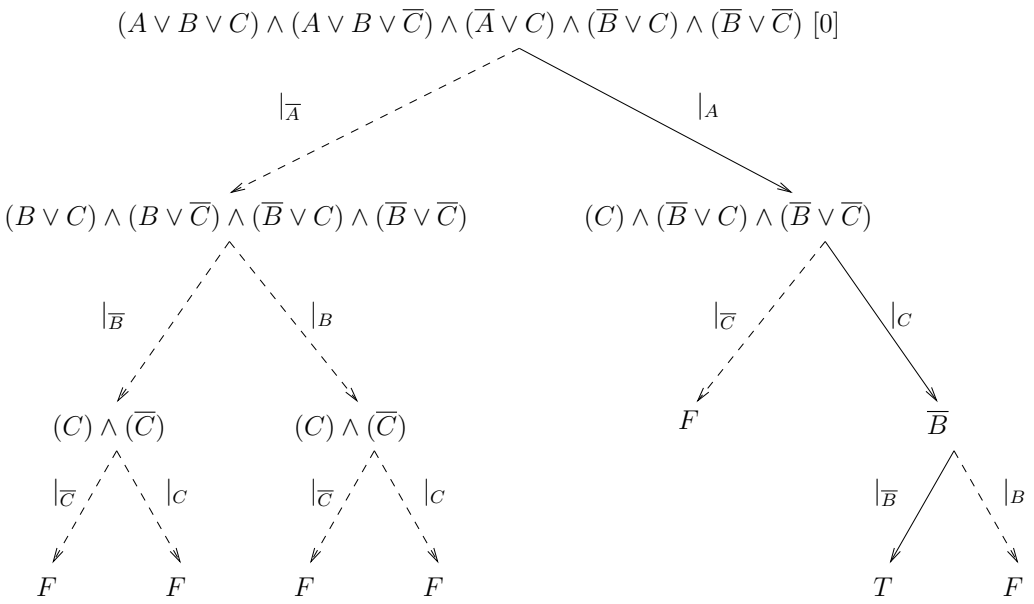

Figure 2. DPLL Binary Tree

other categories they can bring improvements and so we describe them briefly. The final preliminary stage is a static ordering of the variables which is very important to reduce the width of the ternary tree.

3.1 Standard techniques

Unit clause propagation, the pure literal rule and the elimination of tautologies are applied to the initial proposition. However as the main algorithm executes and tentative assignments are made to variables, new unit clauses and pure literals will arise. Unit clause propagation and the pure literal rule are not applied at that stage.

It is sometimes possible for the clauses of a proposition to be partitioned into disjoint subsets, where no variable appears in the clauses of more than one partition. If this can be done then it is possible to solve each of the independent subsets as a separate proposition. If these sub-problems are all satisfiable the union of their respective satisfying assignments will be a satisfying assignment for the original proposition, whereas if any of the sub-problems is unsatisfiable then the original proposition must have been unsatisfiable. A simple depth-first search starting with an arbitrary variable is used to determine the largest connected subset of the proposition. If this does not cover the input, then a sub-problem has been extracted. This process is repeated until the proposition is covered.

3.2 Variable ordering

An essential characteristic of the `tts` algorithm is that the variables are processed in a static order, determined during the initial phase, and the ternary tree structure (described below) encapsulates this ordering. The performance of `tts` depends critically on this variable ordering. The number of nodes at each level in the ternary tree (which affects the number

and size of the sets of nodes which have to be considered) is effectively the number of *active* clauses, where a clause is active whilst at least one of its variables has been selected, but not all. The number of active clauses can be reduced by ensuring that variables that occur in the same clause are processed near to each other in the ordering.

If the variables are regarded as nodes and the clauses as hyperedges, the number of active clauses is given by the *cut-width* of a hypergraph [11]. The variable ordering problem therefore corresponds to the *minimum linear arrangement (MLA)* problem for hypergraphs and is related to the *min-cut* problem. A perfect solution to this problem is known to be NP-hard, and so an approximation algorithm is used. The fact that the best ordering may not have been found affects the performance but not the correctness of `tts`. The survey by Petit [12] contains a comprehensive analysis of the MLA problem and a range of experimental results are presented which suggest that the best approach to use, when the hypergraph is sufficiently small, is Simulated Annealing[9]. For small benchmarks therefore, (currently < 200 variables and < 350 clauses), `tts` uses a combination of:

1. simulated annealing to provide a good approximations for MLA, but with a significant execution time;

2. a local search to see whether the simulated annealing result can be improved.

These are the ordering methods used when solving all the benchmarks described in Section 6. For larger benchmarks simulated annealing takes too long to produce good arrangements and a faster greedy ordering method is used instead. This enables `tts` to solve some larger benchmarks, but the SAT2007 competition results show clearly that it is not competitive in the *random* and *industrial* categories.

4. The main `tts` Algorithm

Once the variable ordering has been determined the tree building and walking can begin. Calls to the core function represent the traversal of a binary tree in the same way as DPLL but the efficiency arises in a different way. As the partial assignments are made and then analysed, groups of reduced clauses which are found to be unsatisfiable are remembered [2], and if they arise again do not have to be reprocessed.

In DPLL partial assignments are used to simplify the proposition, the process being represented by the operator $|_u$. In `tts` a pre-calculated data structure is used to represent similar simplifications. A ternary tree is created, each node of which represents a proposition. The root represents the complete proposition to be solved, and at each level a variable is eliminated. The children of each node do not however directly correspond to assigning T or F to the current variable. Rather the clauses of the proposition are partitioned into three groups, and the whole proposition is then represented by a set of nodes. The data structure is similar to that of `SATO`, as described by Zhang and Stickel [15], but in `tts` the tree is static once built, overcoming the maintenance overhead criticised by Zhang and Malik [16]. In particular, the fact that the variable ordering is static means that the tree can implicitly represent all permissible partial assignments (i.e. those respecting the ordering) and once it has been constructed no further reference to propositions, clauses or literals is required. The algorithm can proceed purely on the basis of the structure of the tree.

The search process involves carrying forward sets of nodes from one level of the tree to the next, searching for one of the following:

- an empty set of nodes which would represent an empty proposition, i.e. a satisfying assignment has been found;
- a set of nodes containing F which indicates a partial assignment which cannot be extended to a satisfying one, in which case backtracking is required; or
- a set of nodes containing a subset which has previously been found to be unsatisfiable, in which case the search is pruned and again backtracking is required.

4.1 Ternary Tree Building

At the core of `tts` therefore is the ternary tree which represents the proposition. This is an explicit data structure rather than a representation of execution traces. The tree is constructed as follows:

- the root of the tree corresponds to the proposition to be solved.
- each node of the tree has three children, *left*, *middle* and *right* (which are called respectively +, DC (don't care) and - by Zhang).

  The left child represents those clauses from the current proposition that contain the literal u except that the u is removed (where u is the current variable). The right child consists of clauses that contain $\overline{u}$ except that the $\overline{u}$ is removed. The middle child consists of those clauses that contain neither u nor $\overline{u}$. Note that a clause containing both u and $\overline{u}$ would be a tautology and would have been eliminated at an earlier stage.
- As before, when the removal of a u or $\overline{u}$ leaves a clause empty this generates a node corresponding to F. When there are no clauses to be included in a child this generates a node corresponding to T which is not included in subsequent sets of nodes.

This means that, given a node representing a proposition, the partial assignment of the current variable to F can be obtained by combining the left and middle children; the partial assignment to T can be obtained by combining the middle and right children.

We introduce the operators $\overset{u}{\leftarrow}$, $\overset{u}{\rightarrow}$ and $\overset{u}{\downarrow}$ as follows:

$$
\begin{array}{lll}
p \overset{u}{\leftarrow} & \triangleq \{c \backslash \{u\} \cdot c \in p \wedge u \in c\} & \textit{(left)} \\
p \overset{u}{\rightarrow} & \triangleq \{c \backslash \{\overline{u}\} \cdot c \in p \wedge \overline{u} \in c\} & \textit{(right)} \\
p \overset{u}{\downarrow} & \triangleq \{c \cdot c \in p \wedge u \notin c \wedge \overline{u} \notin c\} & \textit{(middle)}
\end{array}
$$

Figure 3 gives the ternary tree for the same proposition as Figure 2. The open-headed arrows ——◇ represent clauses which have been satisfied and don't need to be followed. Again, the trace for the assignment $\{A, \overline{B}, C\}$ has been highlighted using solid lines. Each node has been numbered for later reference.

Unlike the DPLL binary tree, this ternary tree is sufficiently small that it can be built explicitly. The number of levels is, as before, $1 + n$ but the number of F leaves is the same

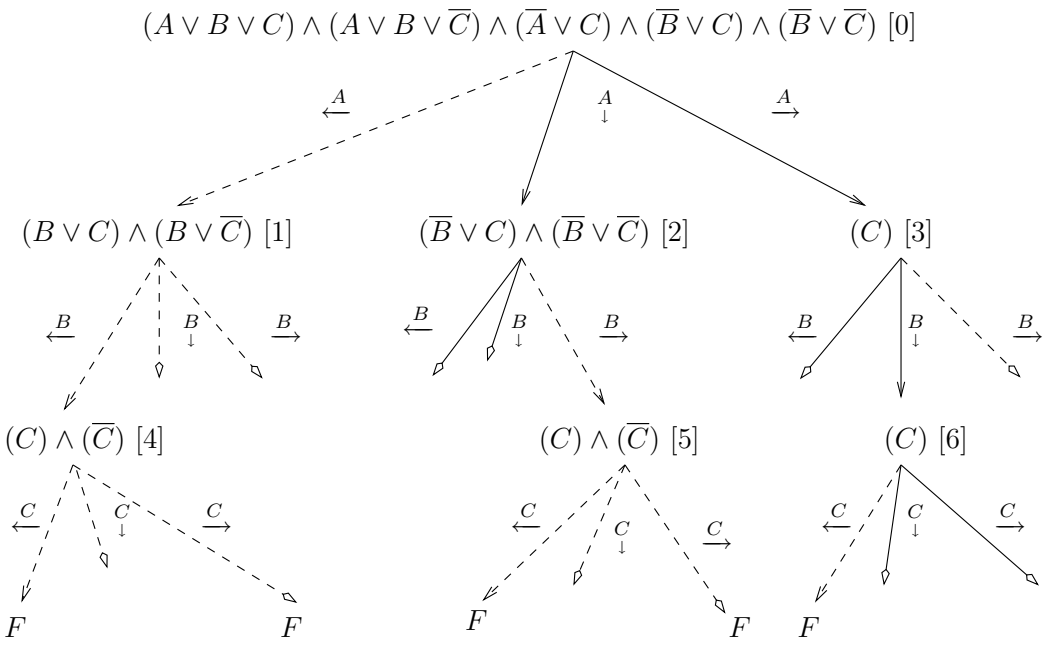

Figure 3. Basic Ternary Tree

as the number of clauses in p. In fact there is a direct correspondence between the F leaves and clauses. The total number of leaves is at most three times the number of F nodes. The tree encodes the $\overset{u}{\leftarrow}$, $\overset{u}{\rightarrow}$ and $\overset{u}{\downarrow}$ operators, meaning that during the tree walking evaluation of these operators is achieved by following tree links.

4.2 Tree Walking

It must be emphasised that although the tree walk is driven by the "small" ternary tree, it nevertheless follows a normal DPLL binary tree pattern with T or F being assigned to a variable at each node.

The traversal does not visit individual nodes but rather sets of nodes from the ternary tree at a time - this set can be interpreted as corresponding to a reduced proposition but following links within the tree is all that is required by the tree walk, no further analysis of the proposition is required. For example, the entire proposition in Figure 3 is represented by the node set {0}. To find the set representing the partial assignment $\overline{A}$ we follow *both* the links $\overset{A}{\leftarrow}$ and $\overset{A}{\downarrow}$ simultaneously, giving the set of nodes {1,2} which corresponds to $(B \vee C) \wedge (B \vee \overline{C}) \wedge (\overline{B} \vee C) \wedge (\overline{B} \vee \overline{C})$ as required. The basic tree walking algorithm is given in Figure 4.

```
boolean tts-0.1 (set of nodes p)
{
    if p = ∅
        return T;
    else if F ∈ p
        return F;
    else
        u = the next variable; (no choice now)
        return tts-0.1(p ←u ∪ p ↓u) or tts-0.1(p →u ∪ p ↓u);
}
```

Figure 4. The basic `tts` algorithm

Using the node numbers on Figure 3 a trace of the sets followed by this algorithm is

$\{0\}$
$\overline{A} \rightarrow \{1,2\}$
$\overline{B} \rightarrow \{4\}$
$\overline{C} \rightarrow \{\mathrm{F}\}$
$C \rightarrow \{\mathrm{F}\}$
$B \rightarrow \{5\}$
$\overline{C} \rightarrow \{\mathrm{F}\}$
$C \rightarrow \{\mathrm{F}\}$
$A \rightarrow \{2,3\}$
$\overline{B} \rightarrow \{6\}$
$\overline{C} \rightarrow \{\mathrm{F}\}$
$C \rightarrow \emptyset$ Success with $A, \overline{B}, C$

4.3 Tree Minimisation

It is apparent in Figure 3 that the sub-trees with roots labelled [4] and [5] are identical and that the operation of the algorithm `tts-0.1` would not be affected if the links pointing to nodes 4 and 5 pointed to the same node. As the tree is constructed a hash table is maintained which maps a set of clauses to the corresponding ternary tree node so that any future attempt to create a node for the same set of clauses can re-use the same node. This leads to the minimised tree in Figure 5.

In fact the data structure minimised in this way is no longer a tree but rather a Directed Acyclic Graph (DAG). It is still referred to as a ternary tree because the difference is not apparent to the tree walking algorithm. This minimisation does not improve the performance of `tts-0.1`, but will be seen to be important when the main version is considered.

4.4 Memoisation

It is quite possible that the function `tts-0.1` will be invoked more than once with the same set of nodes as a parameter. The key to the efficiency of the overall algorithm lies in the

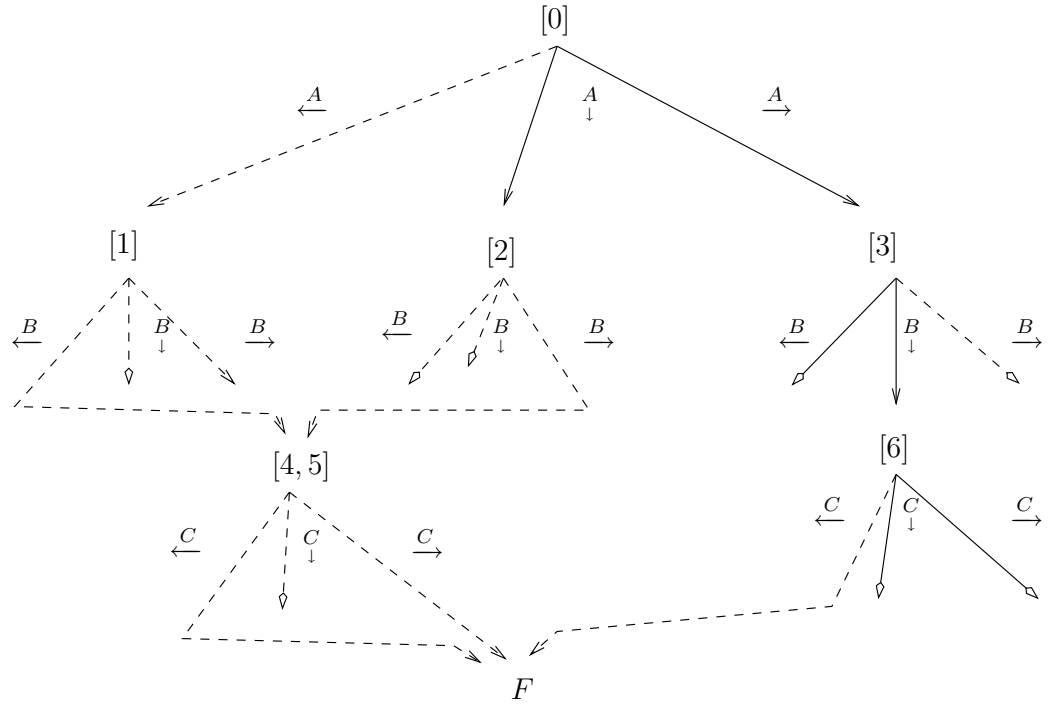

Figure 5. Minimised Ternary Tree

fact that each time `tts-0.1` returns F the input set of nodes is recorded, and if this same set arises again it is known immediately that the required answer is F. In fact the situation is even better than this, in that it may be that not all of the input nodes were required for unsatisfiability.

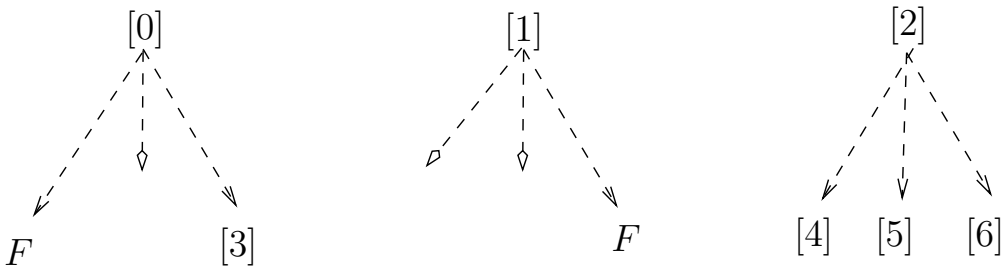

Figure 6. Ternary Tree Fragment

Consider for example the ternary tree fragment Figure 6. Suppose that the proposition p currently under consideration is represented by nodes {0,1,2} and that u is the current variable. We have that

$p \overset{u}{\leftarrow} = \{F, 4\}$
$p \overset{u}{\rightarrow} = \{3, F, 6\}$
$p \overset{u}{\downarrow} = \{5\}$

The next level calls to `tts` will have parameters $p \overset{u}{\leftarrow} \cup p \overset{u}{\downarrow} = \{F, 4, 5\}$ and $p \overset{u}{\rightarrow} \cup p \overset{u}{\downarrow} = \{3, F, 6, 5\}$. In both cases F is included and so backtracking is required, with the current set being recorded as unsatisfiable. In fact node 0 results in F when going left and node 1 results in F when going right so $\{0, 1\}$ is sufficient to lead to F in both cases. $\{0, 1\}$ is recorded as being an unsatisfiable goal rather than $\{0, 1, 2\}$. If any subsequent call to `tts` has $\{0, 1\}$ as a subset of its parameter p then it can return F immediately.

To facilitate this memoisation an output parameter `s` is added to `tts`. If F is being returned, then `s` is set to a subset of `p` which contains those nodes required for the answer F. This leads to the main version algorithm of the algorithm in Figure 7. The tree minimisation described in Section 4.3 can reduce the number of nodes at each level, thereby increasing the number of previously stored sets which can be reused.

```
boolean tts (in set of nodes p, out set of nodes s)
{
    if p = ∅
        return T;
    else if F ∈ p;
        s = {F};
        return F;
    else if ∃s' · s' ⊆ p ∧ unsat(s') (s' already known unsatisfiable)
        s = s';
        return F;
    else
        u = the next variable;
        result = tts(p ←u ∪ p ↓u, s') or tts(p →u ∪ p ↓u, s'');
        if !result
            s = a subset of p covering both s' and s'';
                i.e.  s ←u ∪ ↓u ⊇ s' and s →u ∪ s ↓u ⊇ s''
            record that s is unsatisfiable;
        return result;
}
```

Figure 7. Main version of tts

There is probably scope for improvement in the method currently used for determining $s \subseteq p$ so that $s \overset{u}{\leftarrow} \cup s \overset{u}{\downarrow} \supseteq s'$ and $s \overset{u}{\rightarrow} \cup s \overset{u}{\downarrow} \supseteq s''$. Currently the elements of p are added one by one to an empty set s and the coverage is checked each time until it is complete. It is possible that choosing more carefully the order in which elements are added to s might result in a smaller set being created and subsequent searches being pruned earlier .

The data structure used to store and query these sets is a binary tree derived from the unlimited branching tree (UBTree)[7].

5. Proof of Unsatisfiability

In the SAT2007 competition there was a special track in which only unsatisfiable benchmarks were used and competitors had to give resolution-based proofs that the benchmarks were unsatisfiable. Such a proof derives a series of clauses, starting with the clauses of the input proposition and ending with an empty clause which clearly cannot be satisfied.

Each step eliminates a variable which occurs with opposite signs in two clauses (clashing literals) and creates a new clause consisting of all the other literals in the two clauses. For example, given

$$(A \vee B \vee C) \wedge (\overline{A} \vee C \vee D)$$

we can derive $(B \vee C \vee D)$ by noting the clashing literals A and $\overline{A}$. We write this as

$$(A \vee B \vee C) + (\overline{A} \vee C \vee D) \xrightarrow{\mathsf{res}\, A} (B \vee C \vee D)$$

Thus to certify that

$$(A \vee B) \wedge (\overline{A} \vee B) \wedge (A \vee C) \wedge (\overline{A} \vee C) \wedge (\overline{B} \vee \overline{C})$$

is not satisfiable a possible proof is

$$\begin{array}{lcll} (A \vee B) & + & (\overline{A} \vee B) & \xrightarrow{\mathsf{res}\, A} \quad (B) \\ (A \vee C) & + & (\overline{A} \vee C) & \xrightarrow{\mathsf{res}\, A} \quad (C) \\ (B) & + & (\overline{B} \vee \overline{C}) & \xrightarrow{\mathsf{res}\, B} \quad (\overline{C}) \\ (C) & + & (\overline{C}) & \xrightarrow{\mathsf{res}\, C} \quad () \end{array}$$

It has been found possible to extend the algorithm to generate proofs of this form. In addition to the core algorithm, each call to the revised algorithm `ttsp` returns a clause which is either a clause from the original proposition or has been derived in the permitted manner. It is guaranteed that the clause created does not contain the current variable or any variable which comes after it in the static ordering. This means that the clause created by the initial call to `ttsp` must be empty, which is exactly what is required. Also, it is guaranteed that any variables which do occur in the tree occur with the opposite sign to the corresponding level of the binary tree of calls.

The following cases are used to generate these clauses when `ttsp` is invoked with the node set p. They correspond to the cases within `ttsp` .

- $F \in p$. In this case the original proposition clause leading to this particular occurrence of F in the ternary tree is returned. This cannot contain any variables from later in the static ordering (otherwise the clause would not yet be empty). Also, the current variable must be of the correct sign (or this F would not have been reached).

- One of the clauses returned by the nested calls to `tts` does not contain the current variable. The same clause can be returned at this level as well.

- Both of the clauses returned by nested calls to `tts` contain the current variable. In this case the two occurrences must be of opposite sign and so resolution can be used to create a new clause in which the current variable does not occur.

When memoisation is used, the corresponding clause is stored along with each set of nodes. The revised algorithm `ttsp` is listed in Figure 8.

It should be noted that it is not possible to use the DAG minimisation of the ternary tree prior to invoking `ttsp`. Tree minimisation throws away the information as to which clause led to any particular occurrence of F, and this information is now required. `ttsp` has therefore in general poorer performance than `tts`. It has been noted by van Gelder [10] that `ttsp` produces very good proofs for the pigeon-hole problems but performs very poorly on industrial problems, which is consistent with the performance of `tts`, however (for example) `tts` is able to solve the urquhart benchmarks in [14] while `ttsp` is not.

```
boolean ttsp (in set of nodes p, out set of nodes s, out clause c)
{
    if p = ∅
        return T; /* SAT, so proof irrelevant */
    if F ∈ p;
        s = {F};
        c = clause from which this F derives;
        return F;
    else if ∃s' · s' ⊆ p ∧ unsat(s')
        s = s';
        c = clause stored with s';
        return F;
    else
        u = the next variable;
        result = tts (p ←u ∪ p ↓u, s', c') or tts(p →u ∪ p ↓u, s'',c'');
        if !result
            if u ∉ c'
                c = c';
            else if ū ∉ c''
                c = c'';
            else
                c' + c'' ⟹res u c (derive c by resolution)
            s = a subset of p covering both s' and s'';
            record s,c;
        return result;
}
```

Figure 8. Certified UNSAT version of core `tts` algorithm (`ttsp`)

6. Results

It is well known that the performance of SAT-solvers is very difficult to predict. Solvers which are able to solve benchmarks containing hundreds of thousands of literals can fail on examples with just a few hundred. This is perhaps why the benchmarks in the competitions are divided into three categories and also why a complicated scoring mechanism is used to give some indication of "average, useful" performance.

The approach used here is based on the notion of worst-case performance on a given suite of benchmarks. First we give a brief analysis of the results from Phase 1 of the SAT-2007 competition and use this to justify the creation of a suite of small, difficult benchmarks [14]. The worst-case performances of all the (complete) winners as applied to these benchmarks are then compared.

In order to evaluate the worst-case performance of a solver on a suite of benchmarks, the technique used is to consider a selection of cpu time values and, for each one, to ascertain the smallest benchmark which required the solver to use at least that much time. Using the published results from Stage 1 of the SAT-2007 competition and the times 75, 150, 300, 600 and 1200 seconds yields the results given in Table 1. For clarity, only those solvers which won at least one medal in the `handmade` category are considered for the moment. The solvers are listed alphabetically.

Table 1. Smallest benchmarks requiring at least stated time in SAT Competition Phase 1

Solver	75s	150s	300s	600s	1200s
March_ks	492	516	540	564	588
minisat	516	540	564	588	612
MXC	516	516	540	564	612
SATzilla C	516	540	564	588	612
tts	540	540	**612**	660	660

The highlighted entry for example means that there was a benchmark (in fact it was spence/hard/s97-100) of 612 literals which `tts` took *at least* 300 seconds to solve, but that every benchmark with fewer than 612 literals was solved by `tts` in less time than 300 seconds.

Resolving the time to a finer grain, more detailed results are given graphically in Figure 9. Note that the x-axis, which shows time, has a logarithmic scale. For each solver, a vertex on this graph represents a worst-case pair $< t, n >$ where t is the time taken and n is the number of literals, so that:

- there was a benchmark of size n which took time t to solve; and
- every benchmark of size $\leq n$ took time $\leq t$ to solve

A timeout of 1200 seconds was enforced, and so the rightmost points do not correspond to benchmarks solved in 1200 seconds but rather indicate that at least 1200 seconds were required. Lines are drawn between the vertices only to make the graph clearer, there is no

suggestion that linear interpolation can be used to obtain worst-case execution times for intermediate numbers of literals. Note that increasing in the vertical direction of this graph denotes larger benchmarks for the same execution time and therefore better performance.

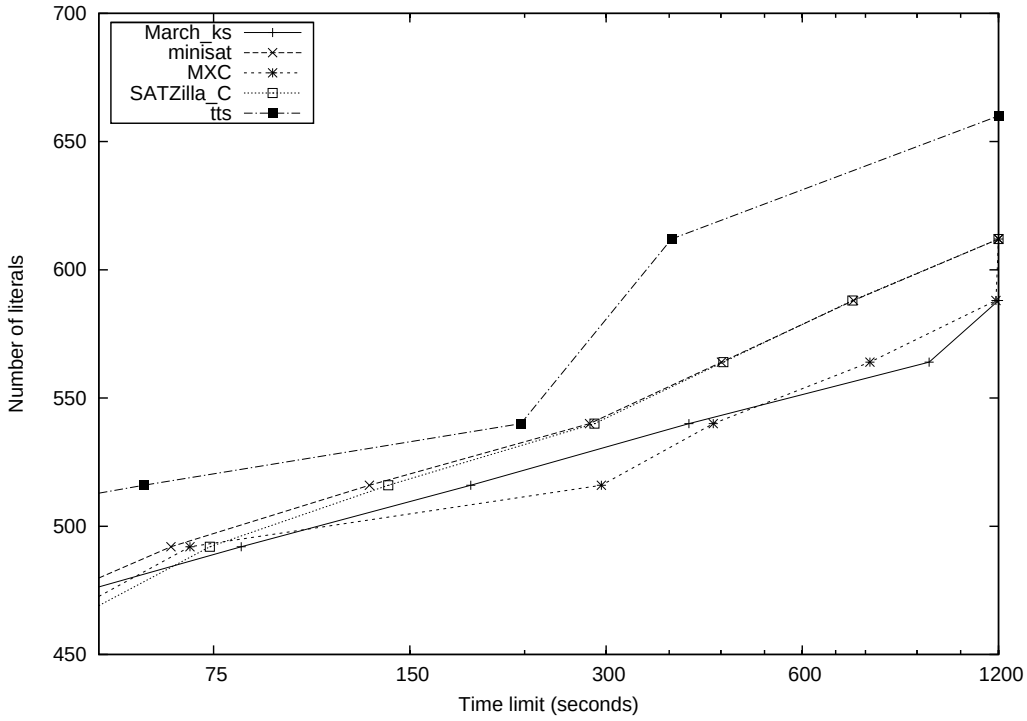

Figure 9. Results from SAT Competition Phase 1

Unfortunately for the purposes of this analysis there were very few benchmarks in the competition which were small enough to contribute. Therefore, a new suite of small, difficult benchmarks (the 3608 benchmarks and the detailed results are available from [14]) has been created by assembling examples from a range of sources, including from these sets all benchmarks containing up to 800 literals. This limit was chosen because all the solvers encountered a benchmark significantly smaller than this which required more than 1200 seconds to solve. In the list below there is a description of the benchmarks within each series which meet this size limit. For the benchmark generators *hgen8* and *sgen* the upper limit on the number of variables was chosen so that the number of literals approached but did not exceed 800 literals, and the total number of benchmarks was gradually increased until the worst-case graphs became reasonably smooth. Benchmarks have been chosen from the following sources:

- From SATLIB[8] - all series containing benchmarks up to 800 literals, that is:
  - Uniform Random 3-SAT (uf and uuf series) - all instances with 20 or 50 variables.
  - Flat graph colouring - all instances with 30 nodes.
  - AIM (artificially generated Random 3-SAT) - the size limit includes benchmarks with 50 and 100 variables.

  - DUBOIS (Randomly generated) - benchmarks with up to 90 variables (`dubois30`).
  - PARITY - up to `par-8-4-c`.
  - II (Inductive inference) - only `ii8a1` falls within the size limit.
  - PHOLE (Pigeon hole) - up to `hole8`.
  - PRET (Encoded 2-colouring) - benchmarks with 60 variables.
  - Bejing (From original SAT competition) - only `2bitcomp_5` falls within the size limit.

- From the Sat-2002 competition[1]
  - Exclusive-or chain - up to `x1_36` and `x2_36`.
  - Urquhart - up to `urquhart3_25.bis`

- Generated by Hirsch's hgen8[6] program using number of variables in the range 100-220 step 10 and random seed in the range 10-90 step 10.

- Generated by the author's sgen[13] program using number of variables in the range 41-129 step 4 and random seed in the range 10-90 step 10.

Some of these benchmarks are quite old and some minor modifications (removing extraneous spaces, removing "%" used as a comment, and removing a final empty clause used to mark the end of the benchmark) were required for them to be accepted by all the solvers. All the medallists in the competition were used in the experiment, and the computer was a 3GHz machine with 2GB of RAM. The results are summarized in Table 2.

Table 2. Smallest benchmarks requiring at least stated time for `sdsb-1-0` benchmarks

Solver	75s	150s	300s	600s	1200s
adaptg2wsat0	239	239	239	239	239
gnovelty+	239	239	239	239	239
kcnfs-2004	516	516	540	564	588
March_ks	516	516	540	564	588
minisat	492	516	540	564	588
MXC	492	492	540	564	564
picosat	420	420	444	444	468
Rsat	444	444	468	468	492
SATzilla C	492	516	540	564	588
SATzilla R	492	516	540	564	588
TiniSatELite C	396	420	420	444	468
tts	540	564	588	612	636

The solvers `adaptg2wsat0` and `gnovelty+` both exceed the time limit of 1200 seconds on the first unsatisfiable benchmarks to be processed - they are listed as being complete solvers

but it appears that their performance is much better on satisfiable rather than unsatisfiable inputs. They are omitted from the graph below. The best six solvers are `kcnfs-2004`, `March_ks`, `MXC`, `Satzilla_C`, `Satzilla_R` and `tts`. As before we now resolve time to a finer grain and the results are given graphically in Figure 10.

It can be seen that `kcnfs-2004`, `March_ks`, `MXC`, `Satzilla_C` and `Satzilla_R` have remarkably similar performances using this metric and that `MXC` is nearly as good, but that `tts` is clearly better. It solves benchmarks with approximately 40 more literals in the same time, or the same benchmarks approximately 2-3 times faster.

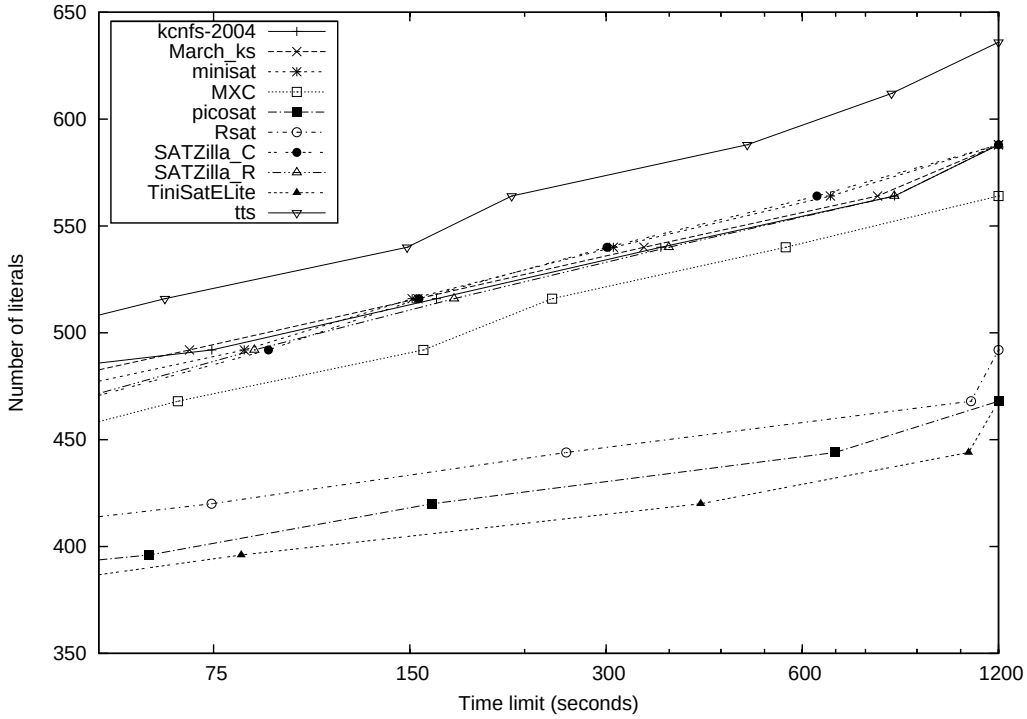

Figure 10. Results for benchmark suite `sdsb-1.0`

It is interesting to note that most of the benchmarks which give rise to the worst-case execution times were generated by `sgen`. This solver was written deliberately to create benchmarks with high bandwidth which would be difficult for `tts`, and it was to be expected then that `tts` would yield its worst performance on these. Some of the generated benchmarks were submitted to the competition and it transpired that other solvers also found them to be very difficult. One of these benchmarks (`s117-100`) was the smallest to remain unsolved at the end of the competition.

7. Conclusions

A solver (`tts`) using a new approach to satisfiability solving has been described and compared with the other medallists from the SAT2007 competition. There are many large benchmarks not presented here which can be solved by other competitors but not by `tts`.

However on what appear to be the worst-case inputs, measuring benchmark size by number of literals, `tts` out-performs the other solvers. A new suite of benchmarks which contains the most difficult known examples has been proposed and made available and again the performance of `tts` on these exceeds that of the other solvers.

It is known from other experiments that although minimising the linear arrangement improves the performance of `tts` in general, it does not necessarily yield the the variable order which gives the lowest possible execution time. At the moment no better metric is known and further research will focus on this area and on improving the choice of an unsatisfiable set of nodes as described in Section 4.4.

8. Acknowledgements

The author wishes to express his gratitude to the reviewers for their careful reading of and helpful suggestions for improvements to the manuscript.

References

[1] Sat competitions. http://www.satcompetition.org/.

[2] Paul Beame, Russell Impagliazzo, Toniann Pitassi, and Nathan Segerlind. Memoization and DPLL: Formula caching proof systems. In *18th IEEE Annual Conference on Computational Complexity*, pages 248–264, July 2003.

[3] Paul Beame, Henry Kautz, and Ashish Sabharwal. Understanding the power of clause learning. *Journal of Artificial Intelligence Research*, 22.

[4] Thierry Castell. Computation of prime implicates and prime implicants by a variant of the Davis and Putnam procedure. In *ICTAI '96: Proceedings of the 8th International Conference on Tools with Artificial Intelligence (ICTAI '96)*, page 428, Washington, DC, USA, 1996. IEEE Computer Society.

[5] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, **5**(7):394–397, 1962.

[6] Edward A. Hirsch. Random generator hgen8 of unsatisfiable formulas in cnf. http://logic.pdmi.ras.ru/~hirsch/benchmarks/hgen8.

[7] Jörg Hoffmann and Jana Koehler. A new method to index and query sets. In *IJCAI '99: Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 462–467, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[8] Holger H. Hoos and Thomas Stützle. Satlib: An online resource for research on sat. In *SAT 2000*, pages 283–292. IOS Press, 2000.

[9] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science, Number 4598, 13 May 1983*, 220, **4598**:671–680, 1983.

[10] Daniel le Berre, Olivier Roussel, and Laurent Simon. The sat'07 contest, the final results. http://www.satcompetition.org/2007/.

[11] Tom Leighton and Satish Rao. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *Journal of the ACM*, **46**(6):787–832, November 1999.

[12] Jordi Petit. Experiments on the minimum linear arrangement problem. *J. Exp. Algorithmics*, 8, 2003.

[13] Ivor Spence. Generator of benchmarks for the satisfiability problem. http://www.cs.qub.ac.uk/~i.spence/sgen.

[14] Ivor Spence. Small, difficult satisfiability benchmarks. http://www.cs.qub.ac.uk/~i.spence/sdsb.

[15] H. Zhang and Mark E. Stickel. Implementing the Davis-Putnam algorithm by tries. Technical report, Iowa City, 1994.

[16] Lintao Zhang and Sharad Malik. The quest for efficient boolean satisfiability solvers. In *CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification*, pages 17–36, London, UK, 2002. Springer-Verlag.